Hash Tables

Hash Tables

Definition

A *hash table* is a data structure used to implement an associative array, a structure that can map keys to values.
A hash table uses a *hash function* to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored.

The implementation of hash tables is called "hashing".
Hashing supports insertions, deletions, and searches in constant ($O(1)$) average time.

- Note that with hash tables, unlike with heaps or binary search trees, items are not ordered.

Hashing

At its most basic form, a hash table is simply an array of fixed size containing the items.

- When dealing with complex items (ie, objects representing person, location, or another compound type) It is often useful to choose one element of the item to serve as the *key*.
  - For example, if the item is the name and address of a person, the key might be a combination of the first three letters of the person's last name and the second number of the street address field.
  - Elmo
    123 Sesame Street
    San Luis Obispo, CA 93407
  - Would have key Elm2.

Hashing

We denote the size of the hash table as *TableSize*.
Each key is mapped into some number from 0 to *TableSize* -1 via a *hash function*.
This hash function should:

- Be simple to compute.
- Map any two distinct keys to different indices.

Collisions

It is best if each key gets assigned to a unique position in the array. However, this is not always possible.

- When two keys generate an identical hash and cause both keys to point to the same index, this is called a *collision*.
- Collisions need to be minimized, but also handled.

When collisions occur there are two simple ways to deal with them:

- separate chaining
- open addressing

Hash Functions

Suppose that the keys to be mapped are the following integers:
32, 37, 43, 45, 48

- Suppose our hash table has size 10 (indexed from 0 to 9).
- Suppose our hash function simply maps key i to $i \mod 10$.
- What would the hash table look like?

Suppose we had the same hash table and hash function, but now our keys are:
27, 37, 48, 57, 78

Hash Functions

Suppose that the keys to be mapped are the following integers:
27, 37, 48, 57, 78

- Suppose our hash table has size 10 (indexed from 0 to 9).
- Suppose our hash function simply maps the tens digit of key i, j to j.
- What would the hash table look like?

Suppose we had the same hash table and hash function, but now our keys are:
232, 37, 743, 245, 48

Hash Functions

Suppose that they keys that are to be mapped are the following strings:
Anna, Bob, Carlee, Annette, Charlie, Stanly, Blake

- How should we hash these keys?
- What would the hash table look like?

Hash Table Sizes

We generally wish the size of the hash table to be prime.
What about the size of the hash table in comparison to the number of elements in the table?

Definition

The *load factor*, λ, of a hash table is the ratio:

$$\frac{\text{the number of elements in the hash table}}{\text{the size of the hash table}}$$

The choice is largely dependent on the application and collision resolution.

Separate Chaining

In this collision resolution strategy, we keep a list (or binary search tree, or new hash table...) of all elements that hash to the same value.
Example:
Suppose that the keys to be mapped are the following integers:
0, 1, 4, 9, 16, 25, 36, 49, 64, and 81
Further suppose that our hash function is simply $x \mod 10$.

- What does the hash table look like if we use separate chaining?

Separate Chaining

Searching:

- Use the hash function to determine which list to traverse.
- Search the appropriate list.

Inserting:

- Check the appropriate list to see whether the element is already there.
- If the element is new, it is inserted into the front of the list.

Separate Chaining

Recall the load factor, λ.

- The average length of a list is λ.
- The time to search is the constant time required to compute the hash function plus the time to traverse the list.
  - In an unsuccessful search, the number of elements to traverse is λ on average.
  - In a successful search, the number of elements to traverse is $1 + \lambda/2$ on average.

When using separate chaining, the load factor should be about 1.

Open Addressing

An alternative to resolving collisions with separate chaining is to try alternative cells until an empty cell is found:
Cells $h_0(x), h_1(x), h_2(x), \ldots$ are tried in succession, where $h_i(x) = (hash(x) + f(i)) \mod TableSize$, with $f(0) = 0$.

- We call f the *conflict resolution strategy*.
- Because all of the data goes directly into the table, our load factor needs to be lower.
  - Ideally, $\lambda < .5$.

Tables that use open addressing are called *probing hash tables*.

Linear Probing

In linear probing, the conflict resolution strategy f is a linear function of i, typically $f(i) = i$.
Example:

- Suppose our keys are $89, 18, 49, 58, 79$.
- Suppose our hash function is $x \mod 10$.
- Suppose we are using linear probing with $f(i) = i$.

Fill in the hash table.

Linear Probing

Even with our small example we saw that areas of the table filled up quickly.
This can occur even if the table is relatively empty.
This phenomenon is called *primary clustering*.

- See text for empirical analysis.

Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem for linear probing.
In quadratic probing, the conflict resolution strategy f is a quadratic function of i, typically $f(i) = i^2$.
Example:

- Suppose our keys are $89, 18, 49, 58, 79$.
- Suppose our hash function is $x \mod 10$.
- Suppose we are using quadratic probing with $f(i) = i^2$.

Fill in the hash table.

Quadratic Probing

In linear probing, performance degraded quickly when the table became full.
In quadratic probing the situation is worse, you may never find an empty cell once the table is more than half full (even earlier if the table size is not prime).

Theorem

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Open Addressing

Can standard deletion be used?
No.

- We must use lazy deletion.

Binary Search Trees vs Hash Tables

Binary Search trees also implement insert and contains operations.

- These are $O(\log n)$ (as opposed to $O(1)$ for hash tables).
- But binary search trees also support order which makes them more powerful.